**PROJECT PHASE 1 DOCUMENTATION**

Functional Simulator for Subset of ARM instruction set

The document describes the design aspect of myRISCSim, a functional simulator for subset of RISCV 32 instruction Input/Output

## Input

Input to the simulator is MEM file that contains the encoded instruction and the corresponding address at which instruction is supposed to be stored, separated by space. For example:

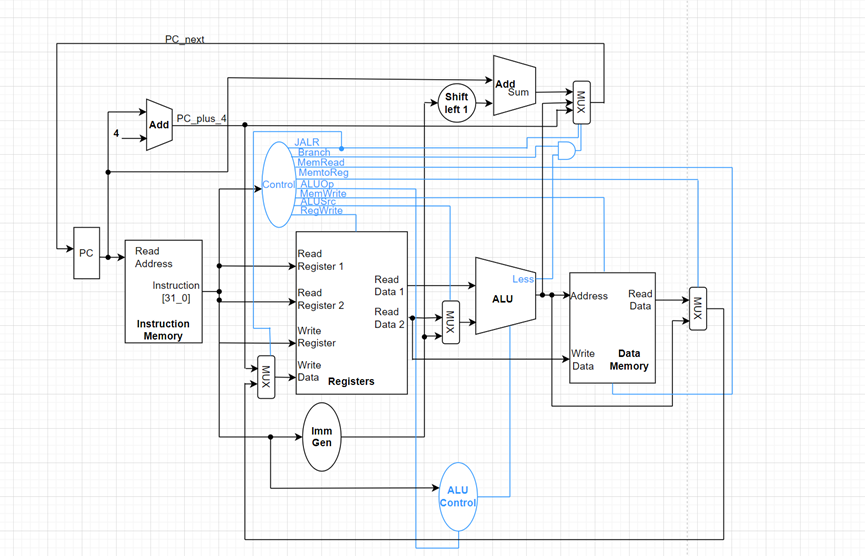
0x0 0x00000093

0x4 0x00100113

0x8 0x00A00193

## Functional Behavior and output

The simulator reads the instruction from instruction memory, decodes the instruction, read the register, execute the operation, access memory if needed, and write back to the register file. The instruction set supported is same as mentioned.

The execution of instruction continues till it reaches last instruction.

# Design of Simulator

## Data structure

Registers, memories, intermediate output for each stage of instruction execution are declared as global static. Being static, the variables are not visible outside the file, thus, make the data encapsulated in the myARMSim.cpp.

There are some variables declared at the beginning of the code. These variables are:

* Register file: A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.
* X: A vector, that is a Register file.
* PC: An integer, holds the current instruction.
* resultALU: An integer, holds the result of execute function.
* resultMEM: An integer, holds data return by memory function or simply we can say return by memory.
* MEM: A vector of unsigned characters representing memory.
* opcode: denotes the operation and format of an instruction.
* instruction word: A static unsigned int, holds the instruction being executed.
* operand1: A static unsigned int, holds the value of the first operand.
* operand2: A static unsigned int, holds the value of the second operand.
* Inst: A bitset representing the current instruction.
* imm: A bitset representing the immediate value of the current instruction.

## Simulator flow:

To design and implement a function simulator for the 32-bit RISC-V ISA instructions, we can follow these steps:

1. Parse the input instructions: The first step is to parse the input instructions and extract the necessary information such as the opcode, source register addresses, immediate values, and destination register addresses.
2. Execute the instructions: The next step is to execute the instructions using the appropriate ALU operation based on the opcode. Some instructions, such as loads and stores, may require accessing memory.
3. Update the register file: After executing an instruction, we update the register file with the result of the operation. We do this in the Register Write-back stage.
4. Forward results: If an instruction in the pipeline requires the result of a previous instruction that has not yet been written back to the register file, we use Result Forwarding to forward the result to the input of the instruction. This avoids stalling the pipeline and improves performance.
5. Repeat for each instruction: We repeat the above steps for each instruction in the input sequence, simulating the behavior of a RISC-V processor executing the program.
6. Return the final register state: After executing all instructions, we return the final state of the register file.

Next we describe the implementation of fetch, decode, execute, memory, and write-back function:-

## *Fetch*

To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction. The instruction is then placed into an instruction register (IR) for decoding.

## *Decode*

The decode stage is responsible for decoding the instruction. The instruction register (IR) holds the fetched instruction. The instruction is decoded to determine the operation to be performed, the operands involved, and the next instruction address. The instruction is decoded using the instruction set architecture (ISA) specification.

## *Execute*

The execute stage is responsible for executing the instruction. The operands are fetched from registers or memory, depending on the instruction type. The ALU performs the operation specified by the instruction. The result is then stored in a register or memory, depending on the instruction type.

## *Memory*

The memory stage is responsible for accessing memory if required by the instruction. If the instruction involves a memory access, the memory address is computed by adding the base address and the offset. The memory access is performed by the data cache. The data is either read from memory or written to memory, depending on the instruction type.

## *Write-Back*

The write-back stage is responsible for writing the result of the instruction back to the register file. The result of the instruction is stored in a register. The register file is updated with the new value. The PC is updated to point to the next instruction based on the control logic.

# Test

We test the simulator with following assembly programs:

* Fibonacci Program
* Sum of the array of N elements.
* Bubble Sort Program.

## Group members:

SUSHIL KUMAR (2021CSB1136)

GYANENDRA MANI(2021CSB1090)

RAKESH MEENA (2021CSB1126)